Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 12 (2012) 212 – 217

Procedia Computer Science

Complex Adaptive Systems, Publication 2
Cihan H. Dagli, Editor in Chief
Conference Organized by Missouri University of Science and Technology
2012- Washington D.C.

Fitting the Problem to the Paradigm: Algorithm Characteristics Required for Effective Use of MapReduce

Fred Highland[a*], John Stephenson[a]

[a]*Lockheed Martin, 3300 Lord Baltimore Dr, Windsor Mill, MD 21244*

Abstract

While the Hadoop MapReduce paradigm offers a linearly scalable approach to solving many complex problems, it does not work for every problem type. General examples of problems that can and cannot be solved with MapReduce have been discussed in a number of sources but the requirements for effective use of MapReduce are not clear. This paper takes the approach that it is not the problem type but the characteristics of the algorithm and the data that must be understood to implement a solution in MapReduce. The paper examines the MapReduce paradigm and derives the key requirements and constraints it implies. These requirements and constraints are stated as a set of rules than can be applied to various problem solutions such that the algorithm and data specifications make effective use of MapReduce. These characteristics can also provide guidance to refactoring incompatible algorithms so that they may be used effectively under MapReduce. Examples of using these characteristics to create effective MapReduce solutions in the healthcare analytics space will be used to illustrate the concepts presented.

Keywords: Big Data, MapReduce, Healthcare Analytics

1. Introduction

The MapReduce paradigm provides a potentially linearly scalable framework that has been used successfully to solve many large, complex problems[1]. However, it does not work for every problem type. The key question is: how to use MapReduce most effectively to solve a problem? Many empirical answers exist in the literature and on the web. Many vendors of big data tools provide examples of solution implementations in MapReduce [2] and there are many examples of problem types [3] for which solutions have been created using MapReduce. But problem types are very general and different algorithms, data and nuances on basic approaches may change the effectiveness of using the MapReduce approach. It has also been addressed by algorithm type [4, 5]. While more specific than general class of problems, the same issues arise in that the application of an algorithm to specific domains and

* Tel.: 301 471 4685.
E-mail address: fred.highland@lmco.com.

 doi:10.1016/j.procs.2012.09.058

problems may make the MapReduce implementation less effective.

Rather than try to classify solutions by high level descriptions, the work presented here determines the algorithm characteristics necessary for effective implementation in MapReduce in order to derive a set of rules to make algorithms and data effective under MapReduce. This is done by performing an analysis of the MapReduce algorithm to derive guidance for solution implementation. The resulting guidance is then used to identify the appropriate solution approach and to structure the approach for maximum effectiveness. It begins with an overview of the MapReduce computational model to identify its key characteristics. It then explores the concepts of algorithm decomposition, data partitioning, data size, and data reduction issues that must be addressed for an effective MapReduce implementation. The application of many of these concepts is illustrated using a complex healthcare analysis problem. Finally a summary of the recommended characteristics and implementation strategies for effective use of MapReduce is presented.

2. MapReduce Model

The basic MapReduce computational model is typically represented as two steps (derived from [1]):

$$\forall(n,m):\ \textbf{map}\ (k_n,v_n) \rightarrow \text{list}(k_m,v_m)$$
$$\textbf{reduce}\ (k_m,\text{list}(v_m)) \rightarrow \text{list}(v_m) \qquad (1)$$

This description is used for consistency with the original work but is somewhat informal. More specifically, MapReduce begins with in input dataset of values (v_n) which is provided to a **map** function. The map function tags the values with keys (k_n), usually derived from the value, and outputs the key-value pairs as a list. The **reduce** function processes each key-value pair in key order, outputting a composite result for each key.

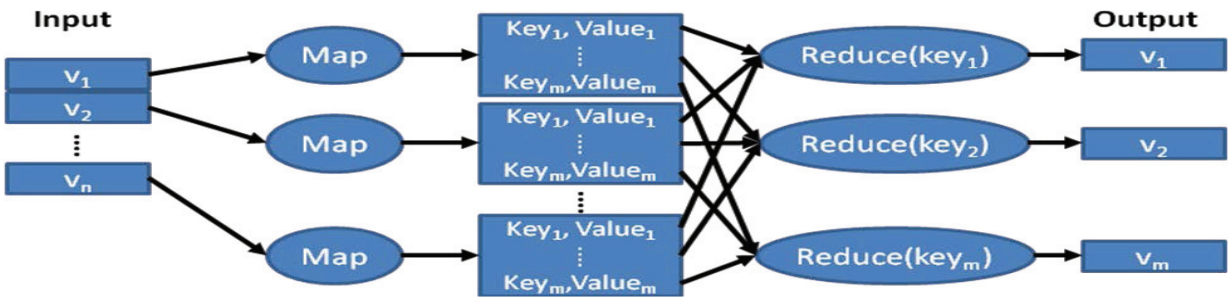

Figure 1 Abstract Implementation of the MapReduce Computational Model

The power of the MapReduce paradigm is not in its simple reduction of the problem structure but in how it is implemented in a computational system. A diagram of that implementation is shown in Figure 1. Following the general model above, the input is divided into value sets $v_1,\ldots v_n$, each containing multiple values. Each v_i set is allocated to a map process on a different virtual processing node. The map processes tag each value in the value set with a key and outputs lists of the resulting key-value pairs (Key_1, $Value_1$,…Key_m,$Value_m$). An output set is created for each map step in the process. MapReduce processing then provides the list of values with the same key to reduce steps on different virtual processing nodes. The reduce steps process all the values producing a "reduced" result. The key computation characteristic of linear scalability is attained from the distribution of processing across multiple virtual nodes which is enabled by partitioning the underlying algorithm using the MapReduce approach.

The MapReduce paradigm implies the following characteristics of the underlying algorithm being implemented.

- **Algorithm is decomposable** – the underlying algorithm can decomposed into a set of steps on subsets of the data that can then be combined to produce the final result. This decomposition is the basis for the definition of reduce (and possibly map) steps.
- **Data is partitionable** – the data can be partitioned into subsets that can be processed independently.
- **Data partition is small enough** – each subset of data used in the initial and intermediate steps is small enough to fit in the storage available on the processing nodes used. The underlying assumption by MapReduce is that this processing can be done on separate virtual processing nodes.

- **Each step reduces data complexity** –complexity (i.e., the number of key-value pairs) is reduced by each step in the process leading to an output of a much smaller number of elements than the original input.

The rationale and implications for each of these assumptions are discussed in more detail below.

3. Algorithm Decomposition

For a given problem to be solved, or algorithm to be implemented, $f(\text{input}) = \text{output}$, the MapReduce relation:

$$\forall(n,m): f(v_n) = \text{list}(v_m) \tag{2}$$

can be decomposed into the MapReduce functions:

$$\begin{aligned}&\forall(n,m): \text{map}\ (k_n,v_n) \rightarrow \text{list}(k_m,v_m)\\ &\quad \text{reduce}\ (k_m,\text{list}(v_m)) \rightarrow \text{list}(v_m)\end{aligned} \tag{3}$$

which composes to:

$$\forall(n,m): \text{reduce}\ (k_m, \text{map}\ (k_n,v_n)) \rightarrow \text{list}(v_m) \tag{4}$$

There are a number of significant implications of this relation.

- The output of each reduce operation for key k_m ($\text{reduce}(k_m,\text{list}(v_m))$) must be an element of the solution.
- The reduce operation for key k_m must compute its output using only k_m and $\text{list}(v_m)$. That is, reduce has no information about other data in the MapReduce computation other than k_m and $\text{list}(v_m)$. This is true both mathematically and physically as other data may present on different virtual processing nodes which may be difficult (expensive) or impossible to access due to network connections.
- The map operation ($\text{map}(k_n,v_n)$) produces a complete list of values with key k_m using only v_n . That is, it has no information about other parts of the MapReduce computation or other $v_i : i<>n$

This implies that the combined map and reduce operations must be a direct decomposition of the desired output function. If the output function produces a list of values, then the MapReduce function must be functions that produce each element or groups of elements of that list. If the output is a single value, then a single reduce function must produce the output, but this may not be efficient.

There are also computational efficiency implications to this. As it is desirable for computation to be distributed across multiple virtual processing nodes (ideally, to attain linear scalability), the number of reduce steps should be large relative to the number of virtual nodes. Generally, if the number of reduce steps (and therefore the number of keys k_m) is greater than the number of virtual processing nodes, then parallelism and distribution of computation can be maximized. It is desirable that the number of reduce steps be much greater than the number of processing nodes for scalability. If the number of reduce steps is significantly less that the number of processing nodes, the computation may not be efficient as all resources are not be used and may become a bottleneck.

The problem has been presented here in its simplest form utilizing a single map-reduce pair. Many interesting problems need to be decomposed into more complex forms consisting of chains of MapReduce steps or:

$$\begin{aligned}&\forall(n,m,s): f(k_n,v_n)) = \text{list}(v_s)\\ &\text{map}_1(k_n,v_n) \rightarrow \text{list}(k_{m1},v_{m1})\\ &\text{reduce}_1(k_{m1},\text{list}(v_{m1})) \rightarrow \text{list}(v_{m1})\\ &\vdots\\ &\text{map}_s(k_{ns},v_{ms-1}) \rightarrow \text{list}(k_{ms},v_{ms})\\ &\text{reduce}_s(k_{ms},\text{list}(v_{ms})) \rightarrow \text{list}(v_s)\end{aligned} \tag{5}$$

The chain of $\text{map}_1\text{reduce}_1..\text{map}_s\text{reduce}_s$ steps represent a more complex version of simple MapReduce with the same properties. It should be noted that there is an implicit dependency between each MapReduce function pair in the sequence requiring that they be executed serially. Therefore the decomposition of the problem into a chain of MapReduce steps does not provide additional opportunities for parallelization within the chain. However, if multiple

map and reduce steps (and therefore keys k_{ns}) can be used at each stage *s*, it provides an opportunity for distribution of processing across the intermediate stages.

The key to effective use of MapReduce is to decompose a function into either a set of separate computations for each output or a series of computations that produce the output, each of which can be performed on part of the input data. This results in a number of independent functions that can be distributed across virtual processing nodes to obtain parallelism and scalability. The number of functions resulting from the decomposition should be kept large, ideally much larger than the number of processing nodes, to maximize effectiveness.

4. Data Partitioning

Understanding the processing of data by MapReduce is important to define the constraints and requirements it places on data. MapReduce is optimized for local data access to minimize data movement and network bandwidth issues. While the underlying scheduling algorithms try to locate computing and data, and more optimal scheduling has been developed [6], the minimization of data movement begins with how it is used by the underlying algorithm. During the initialization of MapReduce processing, the input data is divided into subsets of a user specified size (usually between 16 and 64MB) and these subsets distributed to processing nodes. The map function then processes each record of these subsets, assigning them a key, usually derived from the content of each record. Reduce processing then occurs on the keyed data by processing all key-value pairs with the same key.

This processing has a number of implications on the nature of the data and how it can be processed. First, the data must be structured into records composed of a set of related items and contain information from which a meaningful key value can be derived. Second, the input data must be partitionable into subsets based on these records (MapReduce provides user functions to address this).

As described in the previous section, the data must be grouped, by key values, into disjoint sets for processing by the reduce operations. This forms a hierarchical relationship between subsets of the input data, the key-value pairs produced by the map step, the reduce operations and the output results as shown in Figure 2. The map and reduce functions (or the composition of a chain of map and reduce functions) must use only this data to compute outputs in order for the system to operate efficiently. Maintaining this relationship maximizes parallelism, minimizes data movement and eliminates dependencies among the data optimizing scalability.

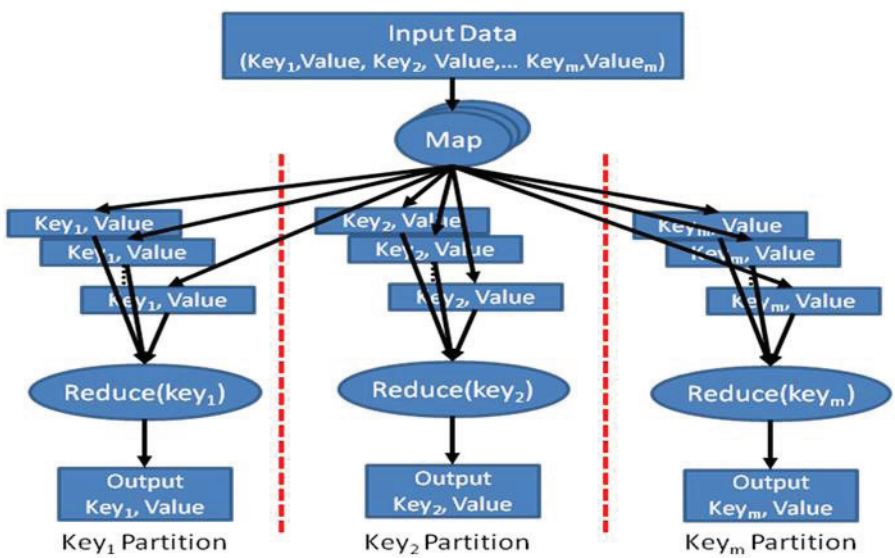

Figure 2 Data Usage by MapReduce Partitions the Data into Disjoint Sets

Real world problems often require more complex data relationships. These can be addressed in various ways with the MapReduce paradigm but care must be taken not to significantly impact the scalability of the implementation. In cases where different input data types are required, multiple map steps, each processing a different input data type,

can be used. In addition, map steps can output multiple key-value pair sets with different data types or structures. These would be processed by subsequent reduce steps with the objective of merging the results into a common key-value pair representing the final result. In cases where the same data is required by multiple reduce steps, it is possible do duplicate data at the map steps. This increases storage usage, communications overhead and can affect scalability and should only be used in isolated cases where the number of records involved is very small. A similar problem arises with the use of global data. If a significant amount of global data is required, MapReduce is not appropriate. However, in small volumes, global data can be used if done judiciously.

5. Data Size and Reduction

MapReduce was designed to operate on clusters of commodity servers and assumes data is moved to the server and utilize high performance local processor, memory and storage resources to optimize performance.

The implications of this are that the data required for each step, particularly the initial map steps, must fit on the available local disk storage. The storage requirements include the input files, output key-values and control information. In addition, MapReduce typically performs multiple map jobs on a single physical node and keeps redundant copies of data for system reliability. The bottom line is that the datasets processed at each map and reduce step need to be of reasonable size, typically 16-64MB or smaller, in order to fit in the resources available at each processing node.

The MapReduce paradigm also assumes the reduction of data complexity at each stage. Under some circumstances, the outputs of the map function may be combined using a built-in combine feature (e.g., if the output is a simple count of the number of items for a key, combine can reduce this to a summary key-value). More typically, the reduce function inputs a list of key-value pairs with the same key value and outputs a much shorter list (often a single element), summarizing the inputs. The use of chained MapReduce approaches can reduce this even further. This approach reduces network traffic and provides for more efficient computation at each level. In general, reduce steps should make order of magnitude reduction (or more) in key-value pairs.

6. Example Application

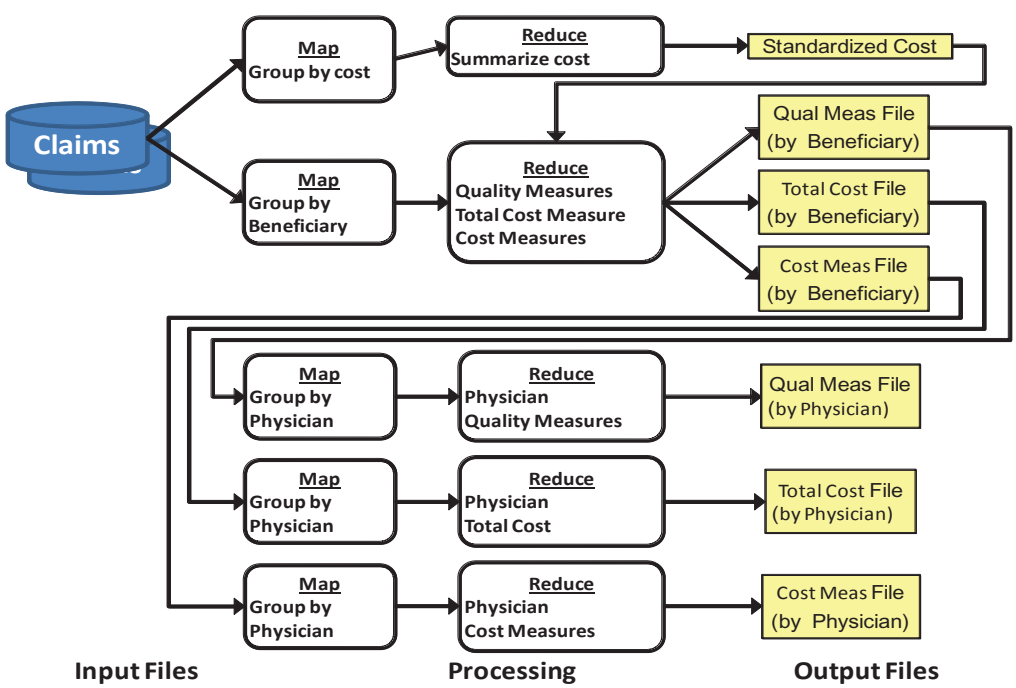

Figure 3 Example of MapReduce Implementation of Healthcare Physician Value Computation

The techniques described here have been used to implement a prototype for a large scale, complex problem in the healthcare domain. As part of the Patient Protection and Affordable Care Act, the Centers for Medicare and Medicaid Services (CMS) are required to compute a payment adjustment value for each physician providing services in 2017. The current approach requires the computation of 63 different quality and utilization measures for almost a million physicians based on data from billions of claims every year. In addition, the number of measures, physicians and complexity of data is expected to continue to grow over time. CMS needs a high performance, flexible and affordable solution to meet this need.

A MapReduce based solution for this problem has been implemented using the design flow in Figure 3. It uses principles put forth in this paper to first partition claims into beneficiary and cost groups. It then computes quality and cost measures from that data by beneficiary and cost type. Finally the measures are aggregated by physician. The partitioning of the data by beneficiary, cost type and the generation of outputs by physician allows the processing to be distributed across multiple processing nodes in small datasets which can be computed on commodity servers. Experiments with the prototype show that it can perform the complex volume of computations required and is linearly scalable for new measures, claims and physicians in the future.

7. Summary and Recommendations

Based on the analysis done here, the algorithms implemented in MapReduce will be optimized if:

- Outputs are a list of values or at least require a list of intermediate values.
- Each value is computable by reduce functions using only a subset of the input data.
- The number of reduce steps is greater than or equal to the number of processing nodes to take advantage of parallelism.
- The data and derived information required to compute the output has a disjoint hierarchical relationship between the inputs and the results. Data duplication and use of global data should be minimized.
- Data required at each Map/Reduce fits within available processor node storage.
- Each major (reduce) step reduces the number of data objects (key-value pairs) by an order of magnitude or more.

The guidance developed here provides a set of requirements for algorithm implementation in MapReduce that is more general than previous work. It has been successfully applied to healthcare analytics problems and is applicable to a wide range of problem types and algorithms.

Acknowledgements

The author would like to acknowledge the other developers on the team: Cathe Reeser, Jermaine Corley and Mike Fink.

References

1. J. Dean and S. Ghemawat, MapReduce: Simplied Data Processing on Large Clusters, OSDI ’04: 6th Symposium on Operating Systems Design and Implementation, USENIX Association (2004) 137
2. Cloudera, Ten Common Hadoopable Problems: Real-World Hadoop Use Cases, http://info.cloudera.com/TenCommonHadoopableProblemsWhitePaper.html, Cloudera Inc., 2011
3. R. Ho, Designing algorithms for Map Reduce, http://horicky.blogspot.com/2010/08/designing-algorithmis-for-map-reduce.html, 2010
4. J. Lin and C. Dyer, Data-Intensive Text Processing with MapReduce, Morgan & Claypool, 2010
5. B. Brumitt, MapReduce Design Patterns, http://www.cs.washington.edu/education/courses/cse490h/11wi/CSE490H_files/mapr-design.pdf, 2008
6. Z. Guo, G. Fox and M. Zhou, Investigation of Data Locality in MapReduce, CCGrid 2012, IEEE Computer Society Press